МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студент гр. 8304 _____ Завражин Д.Г.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2019

Цель работы

Ознакомиться с основными понятиями и приёмами программной реализации иерархических списков, освоить навыки разработки и написания процедур их обработки на языке C++ на примере поставленного задания.

Задание

Вариант 19.

Пусть выражение (логическое, арифметическое, алгебраическое) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ((<операция> <аргументы>)), либо в постфиксной форме (<аргументы> <операция>)). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (* b (- c))) или (OR a (AND b (NOT c))).

В задании даётся один из следующих вариантов требуемого действия с выражением: проверка синтаксической корректности, упрощение (преобразование), вычисление.

Пример упрощения: (+ 0 (* 1 (+ a b))) преобразуется в (+ a b).

В задаче вычисления на входе дополнительно задаётся список значений переменных ((x1 c1) (x2 c2) … (xk ck)), где xi – переменная, а ci – её значение (константа).

В индивидуальном задании указывается: тип выражения (возможно дополнительно - состав операций), вариант действия и форма записи. Всего 9 заданий.

Вариант 19) арифметическое, проверка синтаксической корректности и деления на 0 (простая), постфиксная форма.

Класс *TrivariateHierarchicalList*

С целью программной реализации структуры данных иерархического списка на основе указателей была создана структура данных

TrivariateHierarchicalList, представляющая собой иерархический список, каждый элемент которого хранит либо указатель на подсписок, либо одит из задаваемых шаблоном типов данных.

Интерфейс класса *TrivariateHierarchicalList* состоит из:

- Конструктора, инициализирующего первый элемент списка;
- Деструктора, используемого для освобождения памяти;
- Метода *represent*, возвращающего строковое представление иерархического списка;
- Метода *begin*, возвращающего итератор начала списка;
- Метода *end*, возвращающего итератор конца списка.

Сигнатуры методов класса Expression:

- TrivariateHierarchicalList()
- ~TrivariateHierarchicalList()
- std::string represent()
- Iterator begin()
- Iterator end()

Класс *TrivariateHierarchicalList* имеет два вложенных класса: *Node* и *Iterator*.

Класс *Node* реализует один элемент иерархического списка и написан на основе типа данных *variant* языка C++. Интерфейс класса *Node* состоит из:

- Стандартного конструктора;
- Деструктора, используемого для освобождения памяти;
- Геттера *next* и сеттера *setNext*, обеспечивающих доступ к указателю, указывающему на следующий элемент списка;
- Геттера *content* и сеттера *setContent*, обеспечивающих доступ к содерэимому элемента списка;
- Метода *represent*, возвращающего строковое представление иерархического списка с данным элементом в качестве головного.

Сигнатуры методов класса *Node*:

- ~Node()
- void setNext(Node* const next)

- const std::variant<T, U, V, Node*>& content()
- void setContent(const T& content)
- void setContent(const U& content)
- void setContent(const V& content)
- void setContent(Node* const content)
- std::string represent()

Класс *Iterator* реализует нерекурсивную итерацию по атомарным элементам иерархического списка. Интерфейс класса *Iterator* состоит из:

- Конструктора, запоминающего указатель, указывающий на хранимый в нём элемент и переходящий к первому атомарному после него;
- Унарных операторов инкремента;
- Бинарных операторов равенства и неравенства;
- Метода *getPreviousNodeCount*, возвращающего количество элементов на том же уровне до него.

У него также имеется недоступный извне метод normalizePosition, реализующий переход к следующему атомарному элементу.

Сигнатуры методов класса *Iterator*:

- Iterator(Node *node=nullptr)
- Iterator& operator++()
- Iterator operator++(int)
- Node& operator* ()
- Node* operator-> ()
- bool operator== (const Iterator& that)
- bool operator!= (const Iterator& that)
- size_t getPreviousNodeCount()
- void normalizePosition()

Реализация класса *TrivariateHierarchicalList* и вложенных в него классов, находящаяся в заголовочном файле *trivariatehierarchicallist.h*, приведена вместе со всем исходным кодом программы.

Класс *Expression*

С целью программной реализации структуры данных для программного представления строки данного в условии задания вида на основе иерархического списка была создана структура данных *Expression*, представляющая собой подкласс иерархического списка с добавлениех необходимых методов.

Интерфейс класса, вдобавок к интерфейсу класса *TrivariateHierarchical-List,* интерфейс класса *Expression* состоит из:

- Конструктора, принимающего строку;
- Метода *isCorrect*, проверяющего корректность выражения;
- Метода *getErrors*, возвращающего список ошибок в выражении.

Вдобавок к этому, он имеет следующие недоступныые извне методы:

- Метода *parse*, преобразующего данную строку в иерархический спмсок;
- Метода *checkNodes*, проверяющего проверку вычислимости выражения и деления на 0.

Реализация класса *Expression*, находящаяся в заголовочном файле *expression.h*, приведена вместе со всем исходным кодом программы.

Сигнатуры методов класса *Expression*:

- Expression(const std::string &expression)
- bool isCorrect()
- std::string getErrors()
- std::string::const_iterator parse(ExpressionNode *node, std::string::const_iterator current, std::string::const_iterator end)
- void checkNodes(std::stack<std::variant<T,bool>>& execution_stack)

Вспомогательные функции и типы данных

В процессе выполнения работы были также созданы:

- Класс перечисления *OperationType*, для хранения кодов операции;

- Перегрузка фунции *std::to_string* для *std::string* и класса перечисления *OperationType*;
- Функция *stoT*, приводящая строковый тип данных *std::string* к заданному целочисленному формату.

Их реализация, находящаяся в заголовочном файле *utils.h*, приведена вместе со всем исходным кодом программы.

Функция *main*

Функция *main* выполняет задачу получения от пользователя строки, содержащей анализируемое выражение. Это может происходить двумя способами:

1. Посредством передачи в качестве единственного аргумента командной строки;
2. Посредством ввода по прямому запросу программы.

После получения пути программа передаёт его функции конструктору класса *Expression*, выполняющему вызов его метода parse, обеспечивающего преобразование строки в икрархический список.

Сигнатура функции *main*: int main(int argc, char* argv[]).

Тестирование программы

Тесты, содержащиеся в файле tests.txt, и важные с точки зрения оценки работ программы фрагменты её вывода приведены в таблице 1.

Таблица 1. Тесты, применяемые при тестирование программы.

№	Тест	Обнаруженные ошибки
1		The provided expression is empty
2	()	The provided expression contains empty parentheses
3	(3 a)	There is 1 unused operands left.
4	(5 6 +)	The given expression is correct.
5	(5 6 +))	The string contains the following characters after the last

		closing bracket: ")"
6	(5 6 +) 5	The string contains the following characters after the last closing bracket: " 5"
7	((5 6 +) 5)	There is 1 unused operands left.
8	(5 0 /)	Term 3: Division by 0 encountered.
9	(5 (5 5 -) /)	Term 5: Division by 0 encountered.
10	((6 (5 9 *) /) (4 (6 5 /) -) +)	The given expression is correct.
11	((6 (5 9 *) /) (4 (6 0 /) -) +)	Term 5: Division by 0 encountered.
12	((x1 (x2 x3 *) /) (x4 (x2 (5.5 x3 *) /) -) +)	The given expression is correct.

На всех приведённых выше входных данных программа выдаёт ожидаемый результат; отсюда можно сделать вывод, что данная программа корректно работает во всех охватываемых составленными тестами случаях.

Вывод

В результате выполнания лабораторной работы была реализована программа, отвечающая всем поставленным условиям и проходящая рассмотренное выше составленное в процессе выполнения работы тестирование. Помимо этого, были на практическом примере отточены навыки проектирования, написания и тестирования иерархических списков и алгоритмов работы с ними, владения языком C++.

Исходный код программы

main.cpp

```
#include <iostream>
#include <string>

#include "expression.h"
```

```
using std::cin;
using std::cout;
using std::endl;

using lab2::Expression;

int main(int argc, char* argv[])
{
   if(argc > 2)
   {
      cout << "Too many command line arguments were provided." << endl;
   }

   std::string line;
   if(argc == 2)
   {
         line += std::string(argv[1]);
   }
   else
   {
      cout << "Enter an expression:" << endl;
      std::getline(cin, line);
   }

   Expression<double> expression(line);
   cout << expression.represent() << endl;
   if(expression.isCorrect())
   {
      cout << "The given expression \x1b[1mis\x1b[0m correct." << endl;
```

```
    }
    else
    {
        cout << "The given expression is \x1b[1mnot\x1b[0m correct." << endl;
        cout << expression.getErrors();
    }
    cout << endl; // so, wlile testing, calls would be even somewhat distinct
}
```

utils.h

```
#ifndef LAB2_UTILS_H_
#define LAB2_UTILS_H_

#include <string>

namespace lab2
{
    // enum class OperationType is used to encode an arithmetic operation
     enum class OperationType {ADDITION, SUBTRACTION, MULTIPLICATION,
DIVISION};

    // overload std::to_string for one useful case
    std::string to_string(const std::string& string)
    {
        return string;
    }

    std::string to_string(lab2::OperationType operationType)
    {
        if(operationType == lab2::OperationType::ADDITION)
```

```
        return "+";
    if(operationType == lab2::OperationType::SUBTRACTION)
        return "-";
    if(operationType == lab2::OperationType::MULTIPLICATION)
        return "*";
    if(operationType == lab2::OperationType::DIVISION)
        return "/";
    return "?";
}

std::string to_string(auto value)
{
    return std::to_string(value);
}

// lab2::stoT is a generalization of std::stoi, std::stol, ...
// for a numeric type T
template<class T>
T stoT(std::string);
template<>
short stoT<short>(std::string str){return std::stoi(str);}
template<>
int stoT<int>(std::string str){return std::stoi(str);}
template<>
long stoT<long>(std::string str){return std::stol(str);}
template<>
long long stoT<long long>(std::string str){return std::stoll(str);}
template<>
unsigned short stoT<unsigned short>(std::string str){return std::stoi(str);}
template<>
```

```
    unsigned int stoT<unsigned int>(std::string str){return std::stoul(str);}
    template<>
    unsigned long stoT<unsigned long>(std::string str){return std::stoul(str);}
    template<>
        unsigned long long stoT<unsigned long long>(std::string str){return
std::stoull(str);}
    template<>
    float stoT<float>(std::string str){return std::stof(str);}
    template<>
    double stoT<double>(std::string str){return std::stod(str);}
    template<>
    long double stoT<long double>(std::string str){return std::stold(str);}
}

#endif  // LAB2_UTILS_H_
```

trivariatehierarchicallist.h

```
#ifndef LAB2_TRIVARIATEHIERARCHICALLIST_H_
#define LAB2_TRIVARIATEHIERARCHICALLIST_H_

#include <variant>
#include <stack>

#include "utils.h"

namespace lab2
{
    // TrivariateHierarchicalList class was designed to store either a value of one
    // out of three possible types or a pointer to a sublist
    template<class T, class U, class V>
    class TrivariateHierarchicalList
```

```
{
protected:
    // Node inner class was designed to act as a single node of the
    // TrivariateHierarchicalList class
    class Node
    {
    public:
        Node()
        {
            this->content_ = nullptr;
        }

        ~Node()
        {
            if(std::holds_alternative<Node*>(content_))
                delete std::get<Node*>(content_);
            delete this->next_;
        }

        Node* next()
        {
            return this->next_;
        }

        void setNext(Node* const next)
        {
            this->next_ = next;
        }

        const std::variant<T, U, V, Node*>& content()
```

```
{
    return this->content_;
}

void setContent(const T& content)
{
    this->content_ = content;
}

void setContent(const U& content)
{
    this->content_ = content;
}

void setContent(const V& content)
{
    this->content_ = content;
}

void setContent(Node* const content)
{
    this->content_ = content;
}

std::string represent()
{
    std::string representation = "(";
    auto current = this;
    while(current != nullptr)
    {
```

```
            if(std::holds_alternative<Node*>(current->content_) &&
               std::get<Node*>(current->content()) != nullptr)
                representation +=
                    std::get<Node*>(current->content_)->represent();
            else if(std::holds_alternative<T>(current->content_))
                representation +=
                    lab2::to_string(std::get<T>(current->content_));
            else if(std::holds_alternative<U>(current->content_))
                representation +=
                    lab2::to_string(std::get<U>(current->content_));
            else if(std::holds_alternative<V>(current->content_))
                representation +=
                    lab2::to_string(std::get<V>(current->content_));
            if(current->next_ != nullptr)
                representation += ' ';
            current = current->next_;
        }
        return representation + ')';
    }

private:
    std::variant<T, U, V, Node*> content_;
    Node *next_ = nullptr;
};

// Iterator inner class was designed to facilitate iteration through a
// hierarchical list
class Iterator
{
public:
```

```
Iterator(Node *node=nullptr)
{
   this->current = node;
   this->previousNodeCountStack.push(0);
   this->normalizePosition();
}

Iterator &operator++()
{
   if(this->current == nullptr)
      return *this;
   this->current = this->current->next();
   previousNodeCountStack.top() += 1;
   this->normalizePosition();
   return *this;
}

Iterator  operator++(int)
{
   auto old = *this;
   ++(*this);
   return old;
}

Node &operator* ()
{
   return *(this->current);
}

Node *operator-> ()
```

```
    {
        return &**this;
    }

    bool operator== (const Iterator& that)
    {
        return this->current == that.current &&
                this->nodeStack == that.nodeStack;
    }

    bool operator!= (const Iterator& that)
    {
        return this->current != that.current ||
                this->nodeStack != that.nodeStack;
    }

    size_t getPreviousNodeCount()
    {
        return this->previousNodeCountStack.top();
    }

private:
    std::stack<Node*> nodeStack;
    std::stack<size_t> previousNodeCountStack;
    Node* current = nullptr;

    // ensure that this->current holds a pointer to an atomic node
    void normalizePosition()
    {
        while(this->current == nullptr && this->nodeStack.size() > 0 ||
```

```
            this->current != nullptr &&
            std::holds_alternative<Node*>(this->current->content()) &&
            std::get<Node*>(this->current->content()) != nullptr)
        {
          while(this->current == nullptr && this->nodeStack.size() > 0)
          {
            this->current = this->nodeStack.top()->next();
            this->nodeStack.pop();
            this->previousNodeCountStack.pop();
          }
          while(std::holds_alternative<Node*>(this->current->content()) &&
              std::get<Node*>(this->current->content()) != nullptr)
          {
            this->nodeStack.push(this->current);
            previousNodeCountStack.top() += 1;
            this->previousNodeCountStack.push(0);
            this->current = std::get<Node*>(current->content());
          }
        }
      }
    };

    Node* head()
    {
      return this->head_;
    }

  private:
    Node* head_ = nullptr;
```

```
    public:
        TrivariateHierarchicalList()
        {
            this->head_ = new Node();
        }

        ~TrivariateHierarchicalList()
        {
            delete this->head_;
        }

        std::string represent()
        {
            return this->head_->represent();
        }

        Iterator begin()
        {
            return Iterator(this->head_);
        }

        Iterator end()
        {
            return Iterator();
        }
    };
}
#endif  // LAB2_TRIVARIATEHIERARCHICALLIST_H_
```

expression.h

```
#ifndef LAB2_EXPRESSION_H_
```

```
#define LAB2_EXPRESSION_H_

#include <iostream>
#include <vector>
#include <regex>

#include "trivariatehierarchicallist.h"

constexpr bool DEBUG = true;
namespace lab2
{
   template<class T>
   class Expression
   : public lab2::TrivariateHierarchicalList<T, std::string, lab2::OperationType>
   {
      typedef typename \
         lab2::TrivariateHierarchicalList<T, std::string, lab2::OperationType>::Node \
         ExpressionNode;
   public:
      Expression(const std::string &expression)
      : lab2::TrivariateHierarchicalList<T, std::string, lab2::OperationType>()
      {
         auto expression_ = regex_replace(expression, std::regex("^\\s*"), "");
         expression_ = regex_replace(expression_, std::regex("\\s*$"), "");
         expression_ = regex_replace(expression_, std::regex("\\s+"), " ");
         if(DEBUG)
         {
            std::cout << "Acquired string: \"" <<
                 expression_ << "\"" << std::endl;
            std::cout << "Acquired string length: " <<
```

```
            expression_.length() << std::endl;
    }
    if(expression_.length() == 0 || regex_match(expression_,
                                        std::regex("^\\s*$")))
    {
        this->parsingErrors.push_back("The provided expression is empty");
    }
    else
    {
        this->parse(this->head(), expression_.begin(), expression_.end());
    }
}

bool isCorrect()
{
    std::stack<std::variant<T,bool>> executionStack;
    executionErrors.resize(0);
    this->checkNodes(executionStack);
    if(executionStack.size() > 1)
        this->executionErrors.push_back(std::string("There ") +
            (executionStack.size() > 2 ? "are " : "is ") +
            lab2::to_string(executionStack.size() - 1) +
            " unused operands left.");
    if(this->parsingErrors.size() > 0 || this->executionErrors.size() > 0)
        return false;
    return true;
}

std::string getErrors()
{
```

```
        if(this->parsingErrors.size() == 0 &&
          !(this->executionErrors.size() > 0 || this->isCorrect()))
            return("There are no errors found.\n");

        auto totalLength = 0;
        for(auto error : this->parsingErrors)
            totalLength += error.length() + 1;
        for(auto error : this->executionErrors)
            totalLength += error.length() + 1;

        std::string result;
        for(auto error : this->parsingErrors)
            result += error + "\n";
        for(auto error : this->executionErrors)
            result += error + "\n";
        return "Errors:\n" + result;
    }

private:
    std::vector<std::string> parsingErrors{};
    std::vector<std::string> executionErrors{};

    std::string::const_iterator parse(ExpressionNode *node,
                                      std::string::const_iterator current,
                                      std::string::const_iterator end)
    {
        static size_t depth = 0;
        depth += 1;
        if(DEBUG)
        {
```

```
    std::cout << std::string(depth - 1, ' ') <<
        "|~~~~~~~~~~~" << std::endl;
    std::cout << std::string(depth - 1, ' ') <<
        "| the function \"parse\" was called" << std::endl;
    std::cout << std::string(depth - 1, ' ') <<
        "| depth: " << depth << std::endl;
}

auto currentNode = node;

while(current != end && *current == ' ')
{
    current += 1;
}

if(current != end && *current == '(')
{
    if(depth == 1 && DEBUG)
        std::cout << std::string(depth - 1, ' ') <<
            "| the token \"(\" was acquired: " << std::endl;
    current += 1;
    while(current != end)
    {
        // skip spaces
        while(current != end && *current == ' ')
        {
            current += 1;
        }

        if(current == end)
```

```
{
    this->parsingErrors.push_back("Expression ended unexpectedly");
    return end;
}

// get ")"
if(*current == '(')
{
    if(DEBUG)
    {
        std::cout << std::string(depth - 1, ' ') <<
            "| the token \"(\" was acquired: " << std::endl;
    }
    currentNode->setContent(new ExpressionNode());
    current = parse(
        std::get<ExpressionNode*>(currentNode->content()),
        current, end);
}

// get ")"
else if(*current == ')')
{
    if(DEBUG)
    {
        std::cout << std::string(depth - 1, ' ') <<
            "| the token \")\" was acquired: " << std::endl;
    }
    if(std::holds_alternative<ExpressionNode*>(currentNode->
      content())&& std::get<ExpressionNode*>(currentNode->
      content()) == nullptr)
```

```
    {
        this->parsingErrors.push_back("The provided"\
            "expression contains empty parentheses");
    }
    current += 1;
    break;
}

// get a variable
else if('A' <= *current && *current <= 'Z' ||
                        *current == '_' ||
        'a' <= *current && *current <= 'z')
{
    std::smatch match;
    std::regex regex("^([_A-Za-z][_A-Za-z0-9]*)");
    std::regex_search (current, end, match, regex);
    currentNode->setContent(match[1]);
    if(DEBUG)
    {
        std::cout << std::string(depth - 1, ' ') <<
            "| the token  \"" <<
            std::get<std::string>(currentNode->content()) <<
            "\" was acquired." << std::endl;
    }
    current += match[1].length();
}

// get a number
// fractions of the pattern ".[0-9]+" are not supported
else if('0' <= *current && *current <= '9')
```

```
        {
            std::smatch match;
            std::regex regex("^([0-9]+(\\.[0-9]+)?)");
            std::regex_search (current, end, match, regex);
            currentNode->setContent(lab2::stoT<T>(match[1]));
            if(DEBUG)
            {
                std::cout << std::string(depth - 1, ' ') <<
                    "| the token  \"" <<
                    std::get<T>(currentNode->content()) <<
                    "\" was acquired." << std::endl;
            }
            current += match[1].length();
        }

        // get an operator
        else if(*current == '+' || *current == '-' || *current == '*' ||
                *current == '/')
        {
            if(*current == '+')
                currentNode->setContent(lab2::OperationType::ADDITION);
            else if(*current == '-')
                currentNode->setContent(lab2::OperationType::SUBTRACTION);
            else if(*current == '*')
                                                                currentNode-
>setContent(lab2::OperationType::MULTIPLICATION);
            else if(*current == '/')
                currentNode->setContent(lab2::OperationType::DIVISION);
            current += 1;
            if(DEBUG)
```

```
                {
                    std::cout << std::string(depth - 1, ' ') <<
                        "| the token  \"" << lab2::to_string(std::get<\
                        lab2::OperationType>(currentNode->content())) <<
                        "\" was acquired." << std::endl;
                }
            }

            // ignore other chars
            else
            {
                this->parsingErrors.push_back("Unexpected symbol: " +
                                        lab2::to_string(int(*current)));
                current += 1;
            }

            if(current != end && *current != ')')
            {
                currentNode->setNext(new ExpressionNode());
                currentNode = currentNode->next();
            }
          }
        }
        else
        {
                this->parsingErrors.push_back("The symbol \"(\" is abscent when
necessary");
        }
        depth -= 1;
        if(depth == 0)
```

```
    {
        // report a presence of a character after the last closing bracket
        if(current != end)
        {
            this->parsingErrors.push_back("The string contains the "\
            "following characters after the last closing bracket:\n\t\"" +
            std::string(current, end) + "\"");
        }
    }
    if(DEBUG)
    {
        std::cout << std::string(depth, ' ') << "|~~~~~~~~~~~" << std::endl;
        if(depth > 0)
            std::cout << std::string(depth - 1, ' ') << "| depth: " <<
                depth << std::endl;
    }
    return current;
}

// tries to compute the value of the stored expression in order to find
// all cases of division by 0
void checkNodes(std::stack<std::variant<T,bool>>& executionStack)
{
    size_t termCount = 1;
    for(auto current = this->begin(), end = this->end();
        current != end; termCount += 1, ++current)
    {
        // push a number onto the stack
        if(std::holds_alternative<T>(current->content()))
        {
```

```
            executionStack.push(std::get<T>(current->content()));
            if(DEBUG)
            {
                std::cout << "~~~~~~" << std::endl;
                std::cout << "Term " + lab2::to_string(termCount) +
                    ", the value is "
                    << std::get<T>(current->content())
                    << std::endl;
            }
        }
        // push an unknown value onto the stack
        else if(std::holds_alternative<std::string>(current->content()))
        {
            executionStack.push(true);
            if(DEBUG)
            {
                std::cout << "~~~~~~" << std::endl;
                std::cout << "Term " + lab2::to_string(termCount) +
                    ", the value is undecidable" << std::endl;
            }
        }
        // execute an operation
        else if(std::holds_alternative<lab2::OperationType>(current->content()))
        {
            auto previousNodeCount = current.getPreviousNodeCount();
                    auto operationType = std::get<lab2::OperationType>(current-
>content());
            if(DEBUG)
            {
                std::cout << "~~~~~~" << std::endl;
```

```
    std::cout << "Term " + lab2::to_string(termCount) +
        ", the operation is \"" <<
        lab2::to_string(operationType) + "\""
        << std::endl;
}
// if the operation is called as niladic, report an error
if(previousNodeCount == 0)
{
    this->executionErrors.push_back("Term "
        + lab2::to_string(termCount) + ": The operation \"" +
        lab2::to_string(operationType) +
        "\" has too few (0) arguments.");
}
// if the operation is called as ternary or ..., report an error
else if(previousNodeCount > 2)
{
    this->executionErrors.push_back("Term "
        + lab2::to_string(termCount) + ": The operation \"" +
        lab2::to_string (operationType) + "\" has too many (" +
        lab2::to_string(previousNodeCount) + ") arguments.");
    // remove all its arguments from the stack
    while(previousNodeCount --> 0)
    {
        if(DEBUG)
        {
            std::cout << "Term " + lab2::to_string(termCount) +
                ", the operand " +
                lab2::to_string(previousNodeCount + 1) +
                " is removed from the stack" << std::endl;
        }
```

```
            executionStack.pop();
        }
        // push an unknown value onto the stack as its result
        executionStack.push(true);
        if(DEBUG)
        {
            std::cout << "Term " + lab2::to_string(termCount) +
                ", the result is meaningless" << std::endl;
        }
    }
    // if the operation is called as unary, ...
    else if(previousNodeCount == 1)
    {
        // if it is "-", try to execute it
        if(operationType == lab2::OperationType::SUBTRACTION)
        {
            if(std::holds_alternative<T>(executionStack.top()))
            {
                auto operand = std::get<T>(executionStack.top());
                if(DEBUG)
                {
                    std::cout << "Term "
                        + lab2::to_string(termCount) +
                        ", the operand is " << operand
                        << std::endl;
                }
                executionStack.pop();
                executionStack.push(-operand);
            }
            else
```

```
    {
        if(DEBUG)
        {
            std::cout << "Term "
                + lab2::to_string(termCount) +
                ", the operand is undecidable"
                << std::endl;
        }
        executionStack.pop();
        executionStack.push(true);
    }
}
// if it is not "-", report an error
else
{
    this->executionErrors.push_back("Term "
        + lab2::to_string(termCount) +
        ": The operation \"" +
        lab2::to_string(operationType) +
        "\" has too few (1) arguments.");
    executionStack.pop();
    if(DEBUG)
    {
        std::cout << "Term " + lab2::to_string(termCount) +
            ", the operand " +
            lab2::to_string(previousNodeCount + 1) +
            " is removed from the stack" << std::endl;
    }
    executionStack.push(true);
    if(DEBUG)
```

```
        {
            std::cout << "Term " + lab2::to_string(termCount) +
                ", the result is meaningless"  << std::endl;
        }
    }
}
// if the operation is called as binary, ...
else if(previousNodeCount == 2)
{
    // retrieve the operands in reverse order
    std::variant<T,bool> operand2 = executionStack.top();
    executionStack.pop();
    std::variant<T,bool> operand1 = executionStack.top();
    executionStack.pop();
    if(DEBUG)
    {
        std::cout << "Term " + lab2::to_string(termCount) +
            ", the operand 1 is " + (
            std::holds_alternative<bool>(operand1) ?
            "undecidable" :
            lab2::to_string(std::get<T>(operand1)))
            << std::endl;
        std::cout << "Term " + lab2::to_string(termCount) +
            ", the operand 2 is " + (
            std::holds_alternative<bool>(operand2) ?
            "undecidable" :
            lab2::to_string(std::get<T>(operand2)))
            << std::endl;
    }
    // check for division by 0
```

```
if(std::holds_alternative<T>(operand2) &&
  operationType == lab2::OperationType::DIVISION &&
  std::get<T>(operand2) == 0)
{
   this->executionErrors.push_back("Term "
       + lab2::to_string(termCount) +
       ": Division by 0 encountered.");
   executionStack.push(true);
   if(DEBUG)
   {
      std::cout << "Term " + lab2::to_string(termCount) +
          ", the result is meaningless" << std::endl;
   }
}
// check for undecidable operands
else if(std::holds_alternative<bool>(operand1) ||
       std::holds_alternative<bool>(operand2))
{
   executionStack.push(true);
   if(DEBUG)
   {
      std::cout << "Term " + lab2::to_string(termCount) +
          ", the result is undecidable" << std::endl;
   }
}
// if neither division by 0 nor undecidable operands were
// encountered, execute the operation and save its result
else
{
   T result;
```

```
        if(operationType == lab2::OperationType::ADDITION)
        {
           result = std::get<T>(operand1) +
                    std::get<T>(operand2);
        }
        if(operationType == lab2::OperationType::SUBTRACTION)
        {
           result = std::get<T>(operand1) -
                    std::get<T>(operand2);
        }
        if(operationType == lab2::OperationType::MULTIPLICATION)
        {
           result = std::get<T>(operand1) *
                    std::get<T>(operand2);
        }
        if(operationType == lab2::OperationType::DIVISION)
        {
           result = std::get<T>(operand1) /
                    std::get<T>(operand2);
        }
        executionStack.push(result);
        if(DEBUG)
        {
           std::cout << "Term " + lab2::to_string(termCount) +
               ", the result is " +
               lab2::to_string(result) << std::endl;
        }
     }
  }
  // ignore operands that oppear after an operator
```

```
                if(current->next() != nullptr)
                {
                    while(current->next() != nullptr)
                    {
                        termCount += 1;
                        ++current;
                        this->executionErrors.push_back("Term " +
                            lab2::to_string(termCount) +
                            " comes after an operator, and thus is ignored");
                        if(DEBUG)
                        {
                            std::cout << "~~~~~~" << std::endl;
                            std::cout << "Term " + lab2::to_string(termCount) +
                                ", the value is ignored" << std::endl;
                        }
                    }
                }
            }
            if(DEBUG)
                std::cout << "~~~~~~" << std::endl;
        }
    };
}

#endif  // LAB2_EXPRESSION_H_
```